UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/804,268	03/13/2001	Makoto Muraishi	826.1697/JDH	9108

21171 7590 07/24/2006

STAAS & HALSEY LLP
SUITE 700
1201 NEW YORK AVENUE, N.W.
WASHINGTON, DC 20005

EXAMINER	
CHUONG, TRUC T	
ART UNIT	**PAPER NUMBER**
2179	

DATE MAILED: 07/24/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

Advisory Action Before th Filing of an Appeal Brief	Application No. 09/804,268	Applicant(s) MURAISHI ET AL.	
	Examiner Truc T. Chuong	Art Unit 2179	

--The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

THE REPLY FILED 07 July 2006 FAILS TO PLACE THIS APPLICATION IN CONDITION FOR ALLOWANCE.

1. ☒ The reply was filed after a final rejection, but prior to or on the same day as filing a Notice of Appeal. To avoid abandonment of this application, applicant must timely file one of the following replies: (1) an amendment, affidavit, or other evidence, which places the application in condition for allowance; (2) a Notice of Appeal (with appeal fee) in compliance with 37 CFR 41.31; or (3) a Request for Continued Examination (RCE) in compliance with 37 CFR 1.114. The reply must be filed within one of the following time periods:
   a) ☒ The period for reply expires 3 months from the mailing date of the final rejection.
   b) ☐ The period for reply expires on: (1) the mailing date of this Advisory Action, or (2) the date set forth in the final rejection, whichever is later. In no event, however, will the statutory period for reply expire later than SIX MONTHS from the mailing date of the final rejection.

   Examiner Note: If box 1 is checked, check either box (a) or (b). ONLY CHECK BOX (b) WHEN THE FIRST REPLY WAS FILED WITHIN TWO MONTHS OF THE FINAL REJECTION. See MPEP 706.07(f).

Extensions of time may be obtained under 37 CFR 1.136(a). The date on which the petition under 37 CFR 1.136(a) and the appropriate extension fee have been filed is the date for purposes of determining the period of extension and the corresponding amount of the fee. The appropriate extension fee under 37 CFR 1.17(a) is calculated from: (1) the expiration date of the shortened statutory period for reply originally set in the final Office action; or (2) as set forth in (b) above, if checked. Any reply received by the Office later than three months after the mailing date of the final rejection, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

NOTICE OF APPEAL

2. ☐ The Notice of Appeal was filed on _____. A brief in compliance with 37 CFR 41.37 must be filed within two months of the date of filing the Notice of Appeal (37 CFR 41.37(a)), or any extension thereof (37 CFR 41.37(e)), to avoid dismissal of the appeal. Since a Notice of Appeal has been filed, any reply must be filed within the time period set forth in 37 CFR 41.37(a).

AMENDMENTS

3. ☐ The proposed amendment(s) filed after a final rejection, but prior to the date of filing a brief, will not be entered because
   (a) ☐ They raise new issues that would require further consideration and/or search (see NOTE below);
   (b) ☐ They raise the issue of new matter (see NOTE below);
   (c) ☐ They are not deemed to place the application in better form for appeal by materially reducing or simplifying the issues for appeal; and/or
   (d) ☐ They present additional claims without canceling a corresponding number of finally rejected claims.

   NOTE: _____. (See 37 CFR 1.116 and 41.33(a)).
4. ☐ The amendments are not in compliance with 37 CFR 1.121. See attached Notice of Non-Compliant Amendment (PTOL-324).
5. ☐ Applicant's reply has overcome the following rejection(s): _____.
6. ☐ Newly proposed or amended claim(s) _____ would be allowable if submitted in a separate, timely filed amendment canceling the non-allowable claim(s).
7. ☐ For purposes of appeal, the proposed amendment(s): a) ☐ will not be entered, or b) ☐ will be entered and an explanation of how the new or amended claims would be rejected is provided below or appended.
   The status of the claim(s) is (or will be) as follows:
   Claim(s) allowed: _____.
   Claim(s) objected to: _____.
   Claim(s) rejected: _____.
   Claim(s) withdrawn from consideration: _____.

AFFIDAVIT OR OTHER EVIDENCE

8. ☐ The affidavit or other evidence filed after a final action, but before or on the date of filing a Notice of Appeal will not be entered because applicant failed to provide a showing of good and sufficient reasons why the affidavit or other evidence is necessary and was not earlier presented. See 37 CFR 1.116(e).
9. ☐ The affidavit or other evidence filed after the date of filing a Notice of Appeal, but prior to the date of filing a brief, will not be entered because the affidavit or other evidence failed to overcome all rejections under appeal and/or appellant fails to provide a showing a good and sufficient reasons why it is necessary and was not earlier presented. See 37 CFR 41.33(d)(1).
10. ☐ The affidavit or other evidence is entered. An explanation of the status of the claims after entry is below or attached.

REQUEST FOR RECONSIDERATION/OTHER

11. ☒ The request for reconsideration has been considered but does NOT place the application in condition for allowance because: See Continuation Sheet.
12. ☐ Note the attached Information Disclosure Statement(s). (PTO/SB/08 or PTO-1449) Paper No(s). _____
13. ☒ Other: See attached documents.

WEILUN LO
SUPERVISORY PATENT EXAMINER

Continuation of 11. does NOT place the application in condition for allowance because: From the provided JUnit documents or web site www.junit.org, the document on page 23 clearly shows the publication date was February, 2000; moreover, the Applicants can easily find that the same JUnit document including features and functions of JUnit as presented in the final rejection if following the link http://www.junit.org/news/index.htm?start=121 of the same web site, which is clearly stated that Jtest with JUnit has been available since January 01, 2000 (See attached documents).

JUnit.org

Topics
Articles
Extensions
IDEs
Books
Awards

News

Newer news Older news

The Saboteur Data bug pattern

When a program crashes due to corrupted data, the saboteur can be elusive. Often a program can crash dead in its tracks while manipulating its own internal data, even after working flawlessly for long periods. This article discusses a bug pattern that can be the culprit of this sort of crash, why it exists, and several methods to eliminate it -- before and after it occurs.
Eric E. Allen , May 01, 2001

Diagnosing Java Code: The Liar View bug pattern

GUIs are generally designed with a model-view-controller architecture in which the view is decoupled from the model. The separation presents a challenge to automated testing because it's difficult to verify that a state change in the model is reflected appropriately in the view -- it spawns the infamous "Liar View." This installment of Diagnosing Java Code examines the Liar View bug pattern.
Eric E. Allen, April 01, 2001

XPath Explorer

Why would I want to use XPath Explorer? Maybe you're trying to write an XSL stylesheet and you're tearing your hair out because a complicated XPath matching expression doesn't match what you think it should, and you need some debugging help. Maybe you're using HTTPUnit to unit test your Web site, and you're sick of using the W3C DOM classes to painstakingly walk down your DOM tree. You can use XPath to jump immediately to the value you're looking for and assert that it's present. Using Jaxen, that's as easy as...
Alex Chaffee, March 21, 2002

Stop Over-Engineering!

Patterns are a cornerstone of object-oriented design, while test-first programming and merciless refactoring are cornerstones of evolutionary design. To stop over- or under-engineering, balance these practices and evolve only what you need.
Joshua Kerievsky, Industrial Logic, April 01, 2002

Implement Design by Contract for Java using dynamic proxies

The Design by Contract (DBC) theory can dramatically raise software quality and reusability levels by forcing you to think in terms of contracts. Contracts formally specify the responsibility relationship between a client (class user) and a supplier (class). Additionally, DBC clearly separates specification (what) from implementation (how). This article explains DBC's importance in object-oriented development and describes a DBCProxy framework that achieves DBC transparently in Java using dynamic proxy classes.
Anders Eliasson , February 15, 2002

Discover Your Inner Classes
Unit testing driving you nuts? Learn how using inner classes with the JUnit testing framework can bring you inner peace
Harris W. Kirk , January 01, 2002

XMLUnit
XMLUnit extends JUnit to simplify unit testing of XML. It compares a control XML document to a test document or the result of a transformation, validates documents against a DTD, and (from v0.5) compares the results of XPath expressions.
XMLUnit, March 19, 2001

Jemmy Module
Jemmy is a Java library that is used to create automated tests for Java GUI applications. It contains methods to reproduce all user actions which can performed on swing components (i.e. button pushing, text typing, tree node expanding, ...). JemmyTest is a program written in Java which uses the Jemmy API to test applications. Jemmy is a NetBeans independent module, you can use it separately as well as together with the NetBeans IDE.
NetBeans, February 20, 2002

Server Test Case
A very small and simple extension to JUnit for running test cases inside an ejb server. It makes server tests as easy to write and run as other tests. The fact that a test is run in a server is completely transparent, so you can use it to make test suites which consist of both serverside tests and the usual client side tests.
Jon S Bratseth, March 14, 2002

Jtest
Jtest is an automated Java error prevention tool that performs unit testing and static analysis on classes and JSPs. Jtest complements and extends JUnit. Jtest not only runs JUnit test cases, but also automatically designs and executes additional test cases that verify the code and increase test coverage. Moreover, Jtest automatically creates JUnit test class templates into which you can easily enter test cases, and exports all Jtest test cases in JUnit-compatible format. It even enforces best practices for JUnit test classes during static analysis. By using Jtest to automatically prevent Java errors, you can help your organization boost its CMM level and demonstrate ISO 9001 compliance. Read more about using Automating and Improving Java Unit Testing: Using Jtest with JUnit.
Parasoft, January 01, 2000

Newer news Older news

For a Printable Version, click here *(399 KB PDF)*.
Download the latest version of Adobe Acrobat if you do not have a PDF reader.

Automating and Improving Java Unit Testing: Using Jtest with JUnit

This article is available as a PDF file.

News & Events

Necessary to Deliver Secure, Reliable Web Services
Read more.

Parasoft Simplifies Functional Testing with Automated HttpUnit Web Application Test Suites in WebKing 5.5
Read more.

Parasoft .TEST now integrates with Microsoft Visual

PARASOFT
jtest

Automating and Improving Java Unit Testing: Using Jtest with JUnit

PARASOFT

Introduction

JUnit users can automate the test creation process and further boost software reliability—with virtually no additional effort—by using Parasoft Jtest as well as JUnit. Jtest is an automated error prevention tool that complements and extends JUnit. When JUnit users add Jtest to their arsenal of tools, they can:

- Continue to run their existing JUnit test cases.
- Automatically generate new construction and functionality test cases.
- Automatically export Jtest test cases as JUnit test classes, and then add more test cases by modifying the test classes.
- Automatically create JUnit test class templates, and then add more test cases by modifying the templates.
- Automatically perform static analysis.
- Automatically increase and assess code coverage.

Essentially, by using Jtest and JUnit you streamline the unit testing process so that developers can *actually* perform comprehensive unit testing as often as they *intend* to perform comprehensive unit testing. The increased power that Jtest adds helps developers detect more errors in less time and prevent errors from occurring; this, in turn, leads to a shorter development cycle and a more reliable product.

Testing with JUnit Alone vs. Testing with Jtest and JUnit: An Overview

If you had JUnit at your disposal, you would typically begin the unit testing process by completing the following tasks:

1. Determine test case inputs and expected outcomes.
2. Write the foundational JUnit test class (a class that extends Junit.framework.TestCase) for the class under test.
3. Add to that test class methods which express test cases as assertions.
4. Run the tests in JUnit.

After the test finished, JUnit would report failures in the UI.

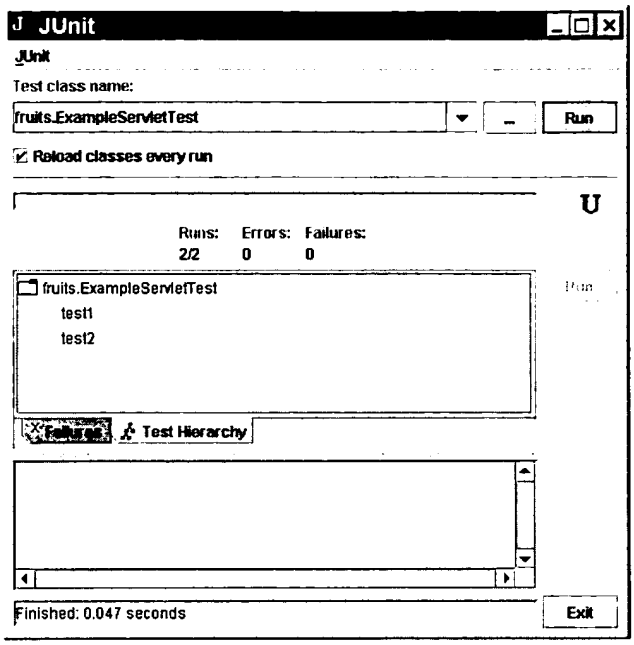

Figure 1: JUnit Results

If you later modified the class, you would need to determine which test cases to add, delete, or modify, then manually recode and extend the test class as needed.

When you have access to both Jtest and JUnit, the unit testing process is facilitated, improved, and expedited. To begin the unit testing process from the Jtest UI, you perform the following steps:

1. Load the class into Jtest.
2. Click Start.

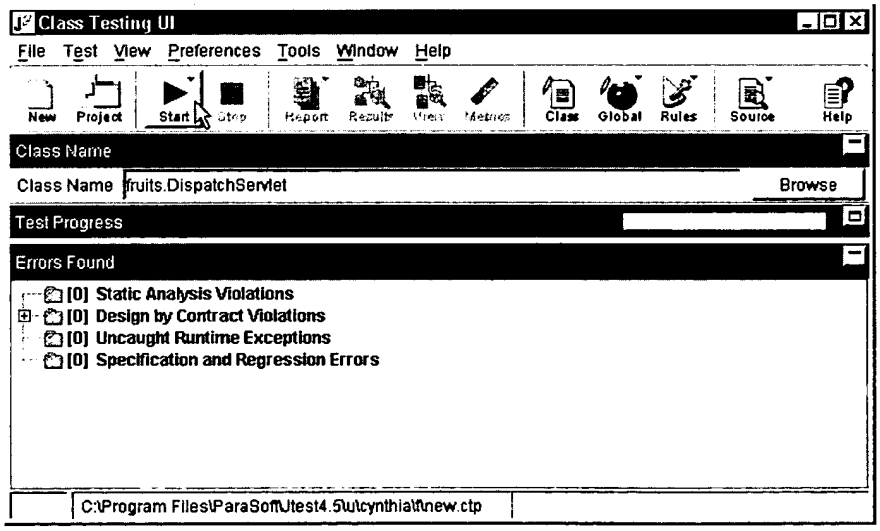

Figure 2: Starting a Unit Test Using Jt st

Jtest will then:

- Examine the class under test.
- Statically analyze the source code.
- Design test cases that can be automatically exported into a JUnit-compatible test class.
- Locate any existing JUnit test cases that appear to be related to this class.
- Run the automatically-generated test cases and any available JUnit test cases.
- Report test results via the UI, HTML or ASCII reports, or XML files that can be transformed into customized reports. These results include graphical coverage reports that show how well the Jtest test cases and JUnit test classes covered the methods of the class under test.

If you wanted to run any or all of Jtest's automatically-generated test cases in the JUnit framework, you could export them as a JUnit-compatible test class, then use that test class in JUnit without having to make any modifications.

You could also create additional test cases for JUnit or Jtest in any of the following ways:

- Automatically create a JUnit-compatible test class template (which includes the required setUp and tearDown methods and contains blank test methods associated with each method of the class under test), then add only the necessary test code.

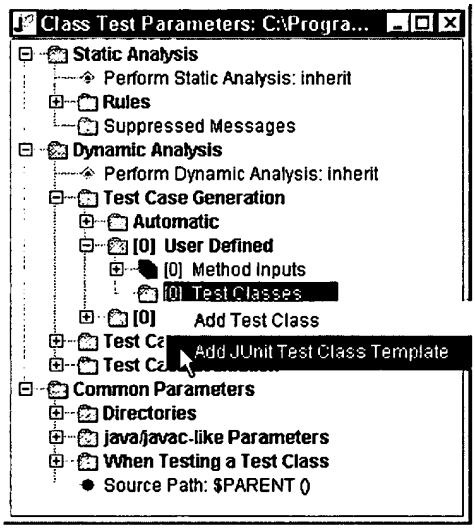

Figure 3: Automatically Creating a Test Class Template

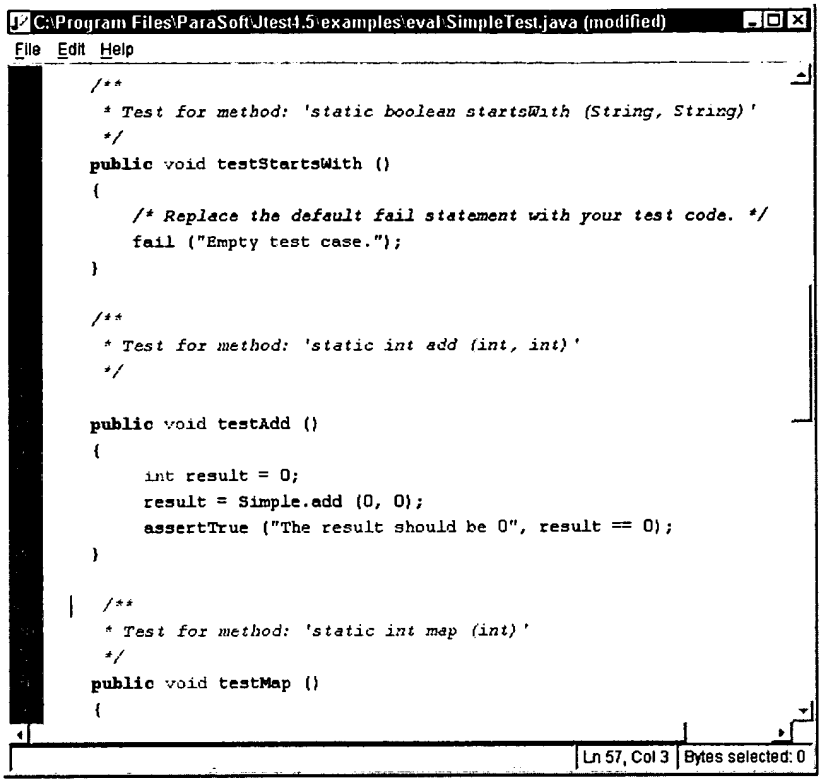

Figure 4: Editing a Test Class Template

- Automatically create a JUnit-compatible test class that represents existing Jtest test cases, then modify the test suite by editing this test class.

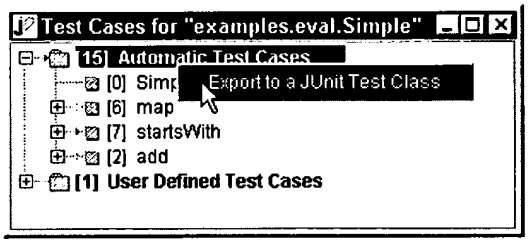

Figure 5: Automatically Exporting Jtest Test Cases as a JUnit Test Class

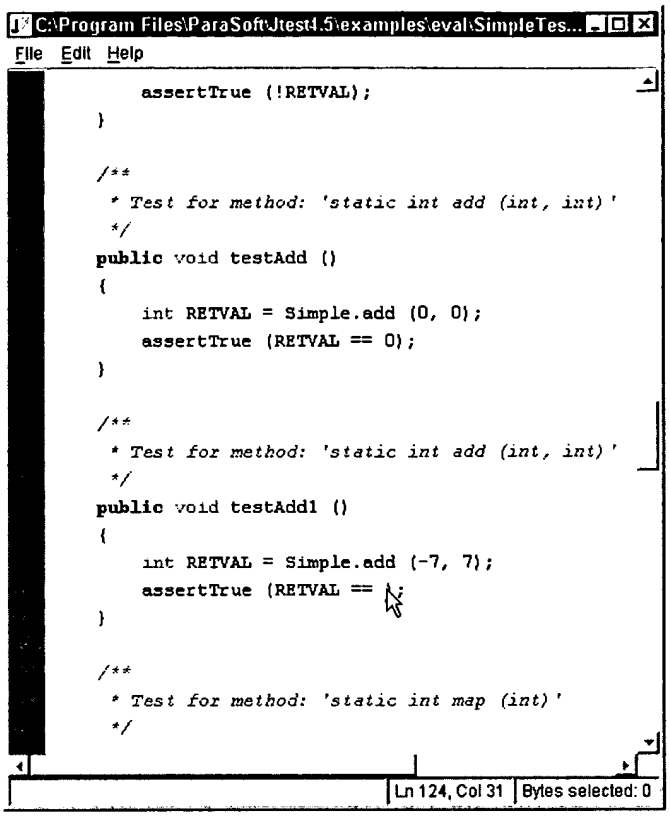

Figure 6: Editing an Exported Test Class

- Modify automatically-generated test cases in the UI.

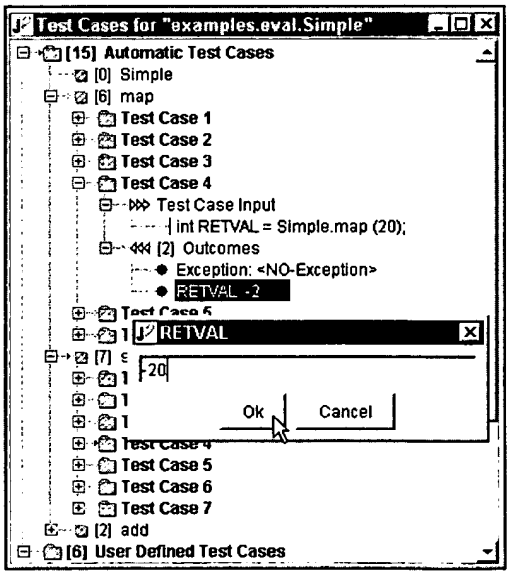

Figure 7: Modifying T st Cases in the Jtest UI

- Add additional test case inputs in the UI.

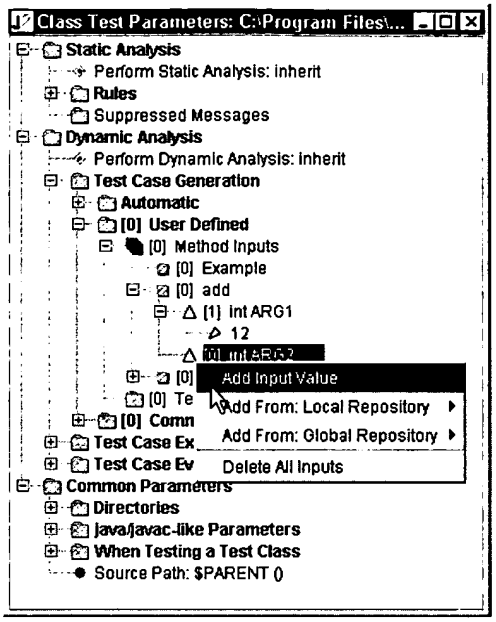

Figure 8: Adding Test Case Inputs in the Jtest UI

If you later modify the class, Jtest automatically updates your test suite during the next test. It creates new test cases to verify the modifications' correctness and robustness, and it repeats the applicable test cases from the earlier test suite. Jtest reports errors if new test cases expose problems, or if the previously-run test cases produce unexpected outcomes.

Benefits of Using Jtest with JUnit

The primary benefit of using both Jtest and JUnit is that by adding Jtest's automatic test case generation and static analysis functionality to the typical JUnit unit testing procedure, development teams gain the ability to efficiently prevent errors. In addition, Jtest's ability to automatically generate test cases, automatically create JUnit test class templates, and automatically export any Jtest test case as a JUnit-compatible test class dramatically reduces the time and effort required to perform thorough unit testing.

Benefits of Automatic Test Case Generation

Developers using both Jtest and JUnit easily improve the quality and speed of the unit testing process. Jtest automatically generates test cases that verify code construction, stability, and functionality. This allows developers to instantly add a battery of additional construction and functionality test cases to their existing JUnit test suite, or to quickly

generate a base set of test cases they can later extend with JUnit test classes (or with test cases they add in the Jtest UI).

This automatic test case generation capability prevents errors in two main ways. First, Jtest creates the volume and range of test cases that is required for thorough construction testing, yet impractical to create manually; this helps developers spot potential weaknesses (for example, instances where the software will fail if it is given unexpected inputs) that might otherwise go unnoticed. Second, Jtest encourages early and frequent testing by allowing developers to test a class with the click of a button; this, in turn, helps the team find and fix problems before someone unwittingly introduces additional errors by adding code that builds upon or interacts with the problematic code.

Another advantage of Jtest's automatic test case generation is that it facilitates faster, more complete regression testing. After a class is modified (for refactoring, redesign, reuse, etc.), Jtest will create new test cases to verify the modifications' correctness and robustness. In addition, Jtest will repeat the applicable test cases from the earlier test suite (including automatically-generated tests); this helps ensure that the modifications do not produce unintended side effects. Developers who write all of their test cases manually would have to spend considerably more time and effort to verify whether modifications introduced problems. They would have to design and code new test cases with each modification, and they would have more trouble exposing undesired side effects because their regression test suite would probably be much less extensive than the one that Jtest could help build.

Finally, the manner in which Jtest automatically generates test cases provides the additional benefit of increased test coverage. Jtest not only generates test cases that test code construction, but also tries to create test cases that execute every possible branch of the method. For example, if the method contains a conditional statement (such as an if block), Jtest will generate test cases that test the true and false outcomes of the if statement. To achieve similar coverage without Jtest, a developer would have to manually write test cases that execute every possible branch of each method of each class; this requires a tremendous amount of work, and is simply not feasible in most situations.

Benefits of Static Analysis

Developers using both Jtest and JUnit also prevent errors by taking advantage of Jtest's flexible static analysis and metrics measurement feature. Jtest statically analyzes each class by parsing its Java source code and comparing it to a set of over 300 industry-respected coding rules designed to identify error-prone code. Available rules prevent such troubles as security problems, performance problems, synchronization problems, portability problems, and internationalization problems; some rules apply to all classes, while others are designed specially for Java technologies such as EJBs and servlets.

Developers can tailor static analysis rules to their projects and create/enforce custom coding rules as needed. The ability to create custom coding standards is a vital tool for effective error prevention: every time an error is detected, developers can determine why the error occurred, develop a custom rule that prevents the same type of error from

reoccurring, then automatically enforce this rule to ensure that any repeat offensives are identified immediately. In addition, developers can use Jtest to automatically measure standard and custom metrics, and to chart metric evolution over the span of a project. This facilitates PSP and related process improvement strategies that help realize the same goal that JUnit promotes: reliable software.

Benefits of JUnit Test Class Support and Generation

Another significant benefit of using both JUnit and Jtest is that Jtest saves developers time by facilitating the creation of additional JUnit test cases which can be executed in Jtest or JUnit. Developers can quickly add user-defined test cases by having Jtest create a JUnit test class template for any class loaded into Jtest, then modifying the template. By automatically creating JUnit test class templates, Jtest relieves developers from having to rewrite the basic JUnit framework for each test class; developers then save time because they only need to focus on adding the specification test code.

Developers using Jtest can further reduce testing time by automatically generating a JUnit-compatible test class representing existing Jtest test cases, then modifying the test class to change existing test cases or add new ones. When developers work in this way, they can easily leverage automatically-generated test cases to verify whether the class is correct. For example, assume that a developer reviewing the results from a previous test run decides to modify some of the construction test cases in order to test a specific set of method inputs. He or she could easily do this by exporting the test cases to a JUnit test class, modifying the test methods to test the necessary specification, and then executing this test class in either JUnit or Jtest. This strategy significantly reduces the amount of code that developers need to write—without sacrificing quality.

How to Use Jtest with JUnit: A Simple Example

The best way to understand how Jtest can boost the power of JUnit is to see how these two tools work together during a test. For a simple example, assume that we have just written Simple.class, a very basic class, and want to verify that it is correct and robust before we start working on the next class.

Simple.java

```
package examples.eval;

public class Simple
{
    public static int map (int index) {
        switch (index) {
        case 0:
        case10:
            return -1;
        case 2:
        case 20:
        default:
            return -2;
        }
    }
```

```
    public static boolean startsWith (String str, String match) {
        for (int i = 0; i < match.length (); ++i)
            if (str.charAt (i) != match.charAt (i))
                return false;
        return true;
    }

    public static int add (int i1, int i2) {
        return i1 + i2;
    }
}
```

If we were relying only on JUnit, we would probably start the testing process by creating a test class that looked something like this:

SimpleTest.java

```
package examples.eval;

import junit.framework.TestCase;

/**
 * SimpleTest is a JUnit test for class examples.eval.Simple
 */
public class SimpleTest
    extends TestCase
{
    /**
     * Constructs a test case for the test specified in the name argument.
     */
    public SimpleTest (String name)
    {
        super (name);

    }

    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    public void testStartsWith ()
    {
        try {
            boolean RETVAL = Simple.startsWith (null, null);
            fail ("A NullPointerException exception should have been thrown.");
        }
        catch (NullPointerException ex) {
            // exception should be thrown
        }
    }

    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    public void testStartsWith1 ()
    {
        boolean RETVAL = Simple.startsWith ("1", "0");
        assertTrue (!RETVAL);
    }

    /**
     * Test for method: 'static int add (int, int)'
     */
```

```
public void testAdd ()
{
    int RETVAL = Simple.add (0, 0);
    assertTrue (RETVAL == 0);
}

/**
 * Test for method: 'static int add (int, int)'
 */
public void testAdd1 ()
{
    int RETVAL = Simple.add (5, 5);
    assertTrue (RETVAL == 10);
}

/**
 * Test for method: 'static int map (int)'
 */
public void testMap ()
{
    int RETVAL = Simple.map (0);
    assertTrue (RETVAL == -1);
}

/**
 * Test for method: 'static int map (int)'
 */
public void testMap1 ()
{
    int RETVAL = Simple.map (2);
    assertTrue (RETVAL == -2);
}

//////////////////

/**
 * Used to set up the test. This method is called by JUnit before each of
 * the tests are executed.
 */
protected void setUp ()
{
    /* Add any necessary initialization code here (e.g., open a socket). */

    // _simple = new Simple ();
}

/**
 * Used to clean up after the test. This method is called by JUnit after
 * each of the tests has been completed.
 */
protected void tearDown ()
{
    /* Add any necessary cleanup code here (e.g., close a socket). */
}

/*
 * Uncomment this variable declaration and add any necessary initialization
 * arguments for it in the setUp() method.
 */
// private Simple _simple;

/**
 * Utility main method. This will run each of the test cases defined in
```

```
     * this class.
     *
     * Usage: java examples.eval.SimpleTest
     */
    public static void main (String[] args)
    {
        /* junit.textui.TestRunner will write the test results to stdout. */
        junit.textui.TestRunner.run (SimpleTest.class);

        /* junit.swingui.TestRunner will display the test results in JUnit's
           swing interface. */
        // junit.swingui.TestRunner.run (SimpleTest.class);
    }
}
```

If we ran this test in JUnit, the test cases would pass, and we would probably check the source code into the source code repository and begin working on the next class.

However, say that we decide to use both Jtest and JUnit. In this case, we prompt Jtest to run the existing JUnit test cases, automatically create new test cases for this class, and statically analyze the class by performing the following steps:

1. Load Simple.class into the Class Testing UI.
2. Open the Class Test Parameters panel to verify that Jtest automatically detected the JUnit test class. (Jtest automatically detects any JUnit test class that is named <name_of_class>Test.class and is located in the same directory as the class under test).

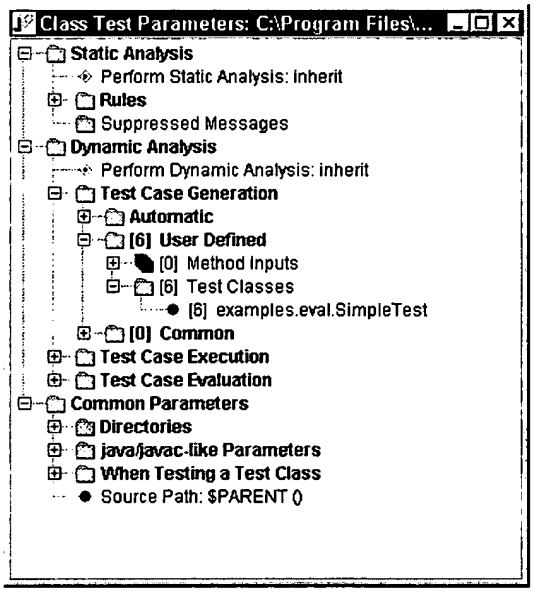

Figure 9: Verifying that Jtest Automatically Detected Our Existing JUnit Test Class

3. Click the Start tool bar button to start the test.

Jtest then examines the class under test, statically analyzes the class, designs test cases that test the class's construction and attempt to cover every branch of every method, and then executes both the automatically-generated test cases and the JUnit test cases. This entire process takes less than 25 seconds. If the class contained Design by Contract comments, Jtest would also design test cases that verified the class's functionality.

When the test is completed, Jtest reports the following results.

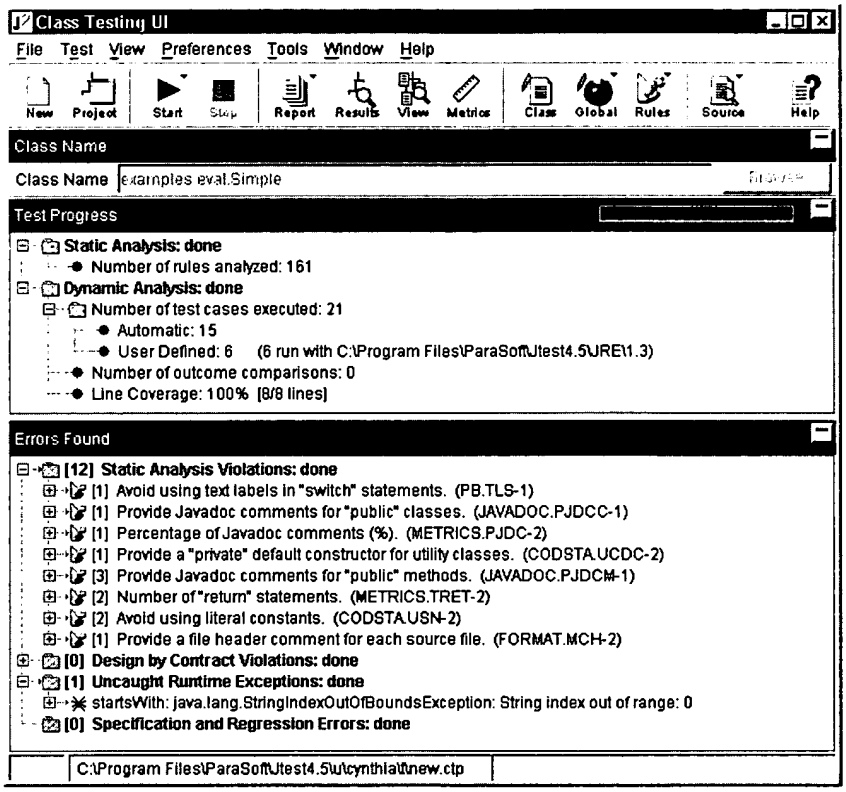

Figure 10: Jtest Test Results

By using Jtest as well as JUnit, we not only verified that the JUnit test cases succeeded, but also identified 12 violations of Java coding guidelines and one uncaught runtime exception. In addition, we automatically generated 15 test cases that can be exported as a JUnit test class and/or used for functionality testing.

To get an idea of the types of weaknesses and problems that Jtest identifies automatically, let's take a quick look at the uncaught runtime exception and one of the static analysis violations that Jtest uncovered.

The uncaught runtime exception message reveals that the startsWith method is implemented incorrectly. The method should return false for the argument "" and "0"; instead, it throws a runtime exception. If the error is not fixed, any application using this class will eventually crash or give incorrect results.

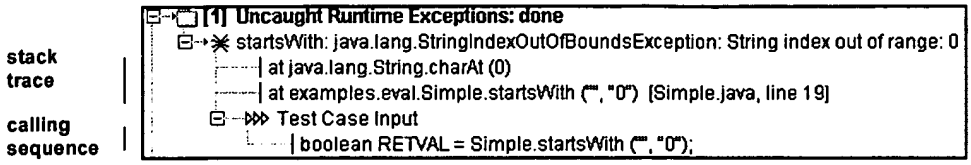

Figure 11: Examining the Uncaught Runtime Exception

The violation of the PB.TLS static analysis rule reveals that the developer of this class inadvertently wrote `case10` instead of `case 10`. If the class is not fixed, it will give incorrect results when it is passed the value 10. If this problem is allowed to remain in the code, it is likely to cause strange, difficult-to-diagnose errors during application testing or in the field

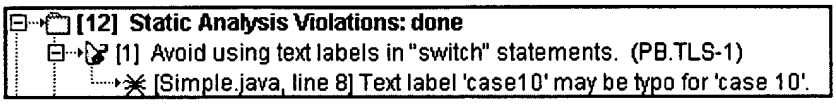

Figure 12: Examining one of the Static Analysis Violations

Next, assume that we want to create a JUnit test class that represents the 15 automatically-generated test cases. If we choose the Export to a JUnit Test Class option, Jtest represents all of the automatically-generated test cases in SimpleTestAuto. We can then modify this test class as much or as little as we want, then run it in either Jtest or JUnit.

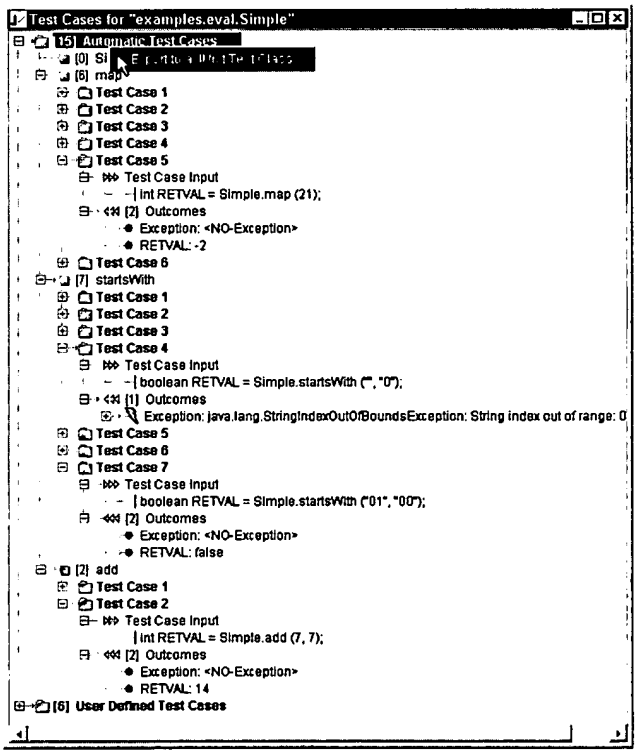

Figure 13: Exporting Automatically-Generated T st Cases as a JUnit Test Class

SimpleTestAuto.java (Excerpts)

```
/*
 * SimpleTestAuto.java
*/
package examples.eval;
import junit.framework.TestCase;
/**
 * SimpleTestAuto is a JUnit test for class examples.eval.Simple
 */
public class SimpleTestAuto
    extends TestCase
{
    /**
     * Constructs a test case for the test specified in the name argument.
     */
    public SimpleTestAuto (String name)
    {
        super (name);
        /*
         * This constructor should not be modified. Any initialization code
         * should be placed in the setUp() method instead.
         */
    }
    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    public void testStartsWith ()
    {
        try {
            boolean RETVAL = Simple.startsWith (null, null);
            fail ("A NullPointerException exception should have been thrown.");
        }
        catch (NullPointerException ex) {
            // exception should be thrown
        }
    }
    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    public void testStartsWith1 ()
    {
```

```
        boolean RETVAL = Simple.startsWith (null, "");
        assertTrue (RETVAL);
    }
    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    /**
     * Test for method: 'static int map (int)'
     */
    public void testMap2 ()
    {
        int RETVAL = Simple.map (10);
        assertTrue (RETVAL == -1);
    }
```

Next, pretend that we never created any JUnit test classes for this class and we now want to add user-defined test cases that verify the class's correctness. Rather than create a JUnit test class from scratch, we could automatically generate a JUnit test class template, then add test code to that template. To create the template, we simply invoke the Add JUnit Class Template feature from the shortcut menu in the Class Test Parameters window.

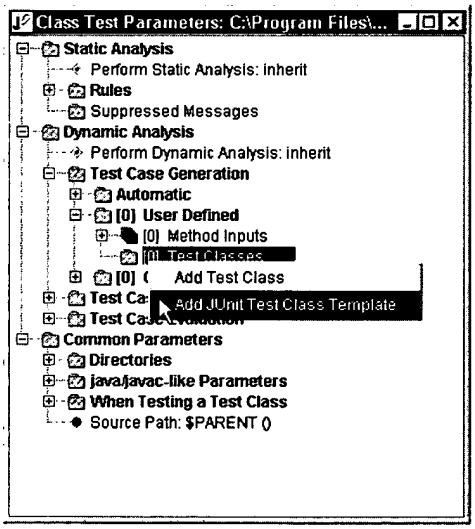

Figure 14: Creating a JUnit Test Class Template

Jtest then produces the following test class template that works in either Jtest or JUnit:

SimpleTest.java (Automatically-Generated Test Class Template)

```
/*
 * SimpleTest.java
 * Created by Jtest on Jun 24, 2002 4:11:10 PM
 */
```

```
package examples.eval;
import junit.framework.TestCase;
/**
 * SimpleTest is a JUnit test for class examples.eval.Simple
 */
public class SimpleTest
    extends TestCase
{
    /**
     * Constructs a test case for the test specified in the name argument.
     */
    public SimpleTest (String name)
    {
        super (name);
        /*
         * This constructor should not be modified. Any initialization code
         * should be placed in the setUp() method instead.
         */
    }
    /**
     * Test for constructor: 'Simple ()'
     */
    public void testSimple ()
    {
        /* Replace the default fail statement with your test code. */
        fail ("Empty test case.");
    }
    /**
     * Test for method: 'static boolean startsWith (String, String)'
     */
    public void testStartsWith ()
    {
        /* Replace the default fail statement with your test code. */
        fail ("Empty test case.");
    }
    /**
     * Test for method: 'static int add (int, int)'
     */
    public void testAdd ()
    {
        /* Replace the default fail statement with your test code. */
```

```
    fail ("Empty test case.");
}
/**
 * Test for method: 'static int map (int)'
 */
public void testMap ()
{
    /* Replace the default fail statement with your test code. */
    fail ("Empty test case.");
}
//////////////////
/**
 * Used to set up the test. This method is called by JUnit before each of
 * the tests are executed.
 */
protected void setUp ()
{
    /* Add any necessary initialization code here (e.g., open a socket). */

    // _simple = new Simple ();
}
/**
 * Used to clean up after the test. This method is called by JUnit after
 * each of the tests has been completed.
 */
protected void tearDown ()
{
    /* Add any necessary cleanup code here (e.g., close a socket). */
}
/*
 * Uncomment this variable declaration and add any necessary initialization
 * arguments for it in the setUp() method.
 */
// private Simple _simple;
/**
 * Utility main method. This will run each of the test cases defined in
 * this class.
 *
 * Usage: java examples.eval.SimpleTest
 */
public static void main (String[] args)
{
```

```
        /* junit.textui.TestRunner will write the test results to stdout. */
        junit.textui.TestRunner.run (SimpleTest.class);
        /* junit.swingui.TestRunner will display the test results in JUnit's
           swing interface. */
        // junit.swingui.TestRunner.run (SimpleTest.class);
    }
}
```

The Test Class created contains a test for each of the methods in the class under test. By default, each test will fail until test code is added. The Test Class also includes comments that describe the purpose of each method, which method the test method is testing, and where users need to modify or add to the code.

For example, if we want to add a test case that verifies whether the Add method produces the expected result (0) for the inputs 0 and 0, we could modify the testAdd method as follows, recompile the test class, then rerun the test in Jtest or JUnit.

```
public void testAdd ()
{
     int result = 0;
     result = Simple.add (0, 0);
     assertTrue ("The result should be 0", result == 0);
}
```

In this case, Jtest's JUnit template generation feature allowed us to create a JUnit test class and test case by writing only 3 lines of code.

Conclusion

These days, almost every Java developer recognizes the value of unit testing and intends to perform comprehensive unit testing as part of his or her regular testing practices. However, unit testing is rarely practiced as early, frequently, or thoroughly as it should be because most developers simply do not have the resources or desire to manually write test case after test case. This problem can be solved by extending JUnit's power with Jtest. When developers use both Jtest and JUnit, unit testing becomes so easy and efficient that developers can realistically integrate it into their daily development practices and reap all of the benefits it has to offer.

This paper provides a very basic introduction to how Jtest can help JUnit users improve software reliability without investing additional time and effort in the development process. To see how Jtest can help your team more efficiently detect and prevent errors in your own software, download Jtest for free at http://www.parasoft.com.